TP n° 6: Filer un mauvais coton.

L'objectif de ce TP est de vous familiariser avec les *thread*s en programmation système.
Le rendu est à faire pour le vendredi 23 mars (avant 23h59) par courrier électronique aux **trois** TDmen [1]. L'objet du mail devra être `[ASR2] Rendu TP06.` [2] Les questions à rendre sont celles où un barême est indiqué. **Le rapport devra faire au maximum 3 pages.** Il doit être au format pdf, et s'intituler `rapport-TP-06-login.pdf`, par exemple `rapport-TP-06-atalon.pdf`.
Tout rendu en retard ou ne respectant pas les consignes (sujet du mail, destinataires, format du rapport...) coûtera 1,5 points sur 10.

Exercice 1. *Oh attention, chérie, ça va couper*

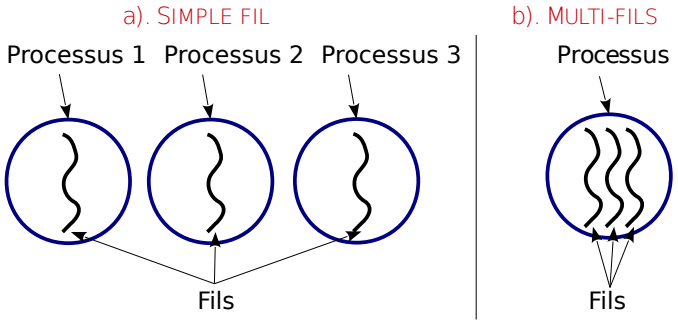

Figure 1 – Illustration du fonctionnement de processus légers, et de processus.

Mise en garde. Avec des `fork`s, vous pouvez risquer de remplir la file de processus du système si vous ne faites pas attention. Sauvegardez bien tout avant de faire quoi que ce soit de dangereux !

Cet exercice a pour but de comparer les performances du déroulement d'un programme qui fonctionne avec des fils ou des processus (voir figure 1).

Pour cela on va effectuer une multiplication matricielle en divisant le travail par blocs. On rappelle que

$$(a_{ij})_{\substack{1\le i\le n\\1\le j\le m}} \cdot (b_{ij})_{\substack{1\le i\le m\\1\le j\le p}} = \left(\sum_{k=1}^{m} a_{ik}\cdot b_{kj}\right)_{\substack{1\le i\le n\\1\le j\le p}}$$

La mémoire sera partagée à l'aide de `mmap` pour la version avec les processus.

1. **(/1)** Écrivez un programme `ex1_matmul` (sans `fork`s ni `pthread`s pour le moment) qui prend en entrée deux matrices sous la forme suivante :

```
2       // Nombre de matrices à suivre (ici toujours 2)
3 3     // taille de la première matrice
1 2 3   // entrées de la première matrice
4 5 6
7 8 9

3 3     // taille de la deuxième matrice
```

1. {valentin.lorentz,remy.grunblatt, alexandre.talon}@ens-lyon.fr
2. sans le point, qui terminait juste la phrase.

```
1 4 7  // entrées de la deuxième matrice
2 5 8
3 6 9
```

(le traitement des commentaires n'est pas demandé) et qui renvoie le produit des deux matrices.
Ici la réponse sera

```
  14   32   50
  32   77  122
  50  122  194
```

2. **(/1)** Écrivez un programme `gen_matrices` qui prend en argument deux entiers n et k et qui tire deux matrices carrées de taille n dont les éléments sont à valeur dans $[-k, k] \cap \mathbb{Z}$ uniformément, et renvoie dans `stdout` le résultat dans le format de matrices donné plus haut.
3. **(/1)** Écrivez la version `ex1_matmul_proc` utilisant un `fork` pour diviser le travail case à case (en utilisant la stratégie maître-esclave vue dans le *TP03*, mais sans signaux).
4. **(/1,5)** Faites de même pour la version `ex1_matmul_fils` qui utilise les `pthread`s.
5. **(/3)** Dans le rapport, comparez les performances de chaque solution. Vous pourrez pour cela utiliser *time*, *gnuplot*, *R*, ou un tableur quelconque. Incluez les courbes dans le rapport.
   (a) En particulier, à partir de quand l'accélération devient sous-linéaire ?
   (b) En particulier, y-a-t-il des instances avec une accélération super-linéaire ?

Exercice 2. *Proberen*

1. **(/1,5)** Compilez le fichier `sum.c` qui calcule la somme d'un vecteur en parallèle.
   (a) Qu'observez-vous ?
   (b) Résolvez le problème.
2. **(/1,5)** Maintenant que le problème est résolu, qu'observez-vous ?
   (a) Essayez d'améliorer la solution.

Exercice 3. *Into the code*
Cet exercice est optionnel.

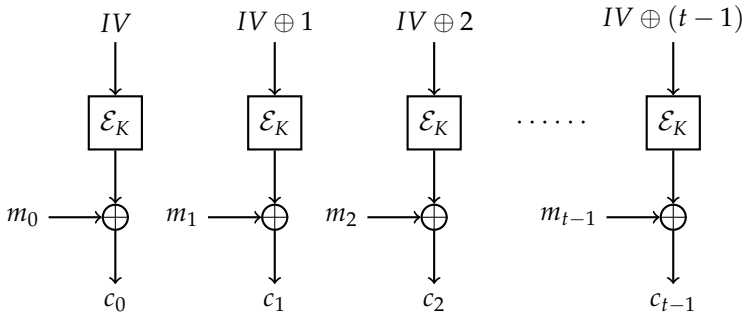

FIGURE 2 – Mode de chiffrement CTR.

Mise en garde : N'utilisez jamais du code que vous avez fait vous-même pour faire de la vraie cryptographie, à moins de *vraiment* savoir ce que vous faites. Ici ce n'est qu'un exemple jouet pour ne pas faire des exercices rébarbatifs et qui se ressemblent tous.

Dans cet exercice on va construire une parallélisation d'un chiffrement par bloc en mode CTR (voir figure 2). L'idée est qu'en général un chiffrement est une fonction $\mathcal{E} : \{0,1\}^k \times \{0,1\}^b \to \{0,1\}^\ell$ qui permet de traiter à chaque étape un message de taille fixée. Pour passer à une taille plus grande, les messages sont chiffrés suivant un mode d'opération (il y en a pleins), qui permet de dire comment chiffrer et déchiffrer des chaînes de bits aussi grandes qu'on veut.
Le mode CTR (CompTeuR) est construit comme décrit en figure 2. Le principe est de partir d'un chiffrement symétrique $\mathcal{E} : (K, m) \in \{0,1\}^k \times \{0,1\}^b \to c \in \{0,1\}^\ell$, où on notera $\mathcal{E}_K = \mathcal{E}(K, \cdot)$, de diviser notre message $M \in \{0,1\}^{t\times\ell}$ en t blocs de taille $\ell : m_0, \ldots, m_{t-1}$ de chiffrer un compteur à partir d'une valeur tirée uniformément appelée « vecteur d'initialisation » (ou IV en anglais), et on utilise la sortie du schéma de chiffrement comme un masque jetable pour chiffrer les différents blocs : $\forall i \in \{0, 1, \ldots, t-1\}, c_i = m_i \oplus \mathcal{E}_K(IV \oplus i)$, et le message chiffré final consistera en $(IV, (c_i)_{0\leq i<t})$
Pour cela on va utiliser notre programme précédent pour la multiplication matricielle sur les threads pour construire un cryptosystème symétrique construit sur le problème **LWE** pour *learning with errors*.

Definition (Learning with errors)**.** On note $\mathbb{M} = \mathbb{Z}_{/q\mathbb{Z}}$ avec $q = 2^k$. Étant donné une matrice $\mathbf{A} \in \mathbb{M}^{m\times n}$ avec $m > n$, un vecteur $\mathbf{s} \in \mathbb{M}^n$ et $\mathbf{e} \in \mathbb{Z}^m$ tel que $\|\mathbf{e}\|_\infty < \mathbf{q/4}$, on appelle distribution LWE la distribution $(\mathbf{A}, \mathbf{A}\cdot\mathbf{s} + \mathbf{e})_{\mathbf{s},\mathbf{e}} \in \mathbb{M}^{m\times n} \times \mathbb{M}^m$.
L'hypothèse LWE dit qu'il est difficile de distinguer la distribution LWE de la distribution uniforme sur $\mathbb{M}^{m\times n} \times \mathbb{M}^m$ pour des paramètres m, n, k bien choisis.

Dans la suite nous prendrons $m = 256$, $n = 100$, $k = 60$ (pour que ça rentre dans un mot machine).
Pour chiffrer, on utilise le bloc suivant, appelé $\mathbf{LWE}_{\text{sym}}$:

Keygen() : renvoie la clef secrète $\mathbf{s} \in \mathcal{U}(\mathbb{M}^n)$ et une matrice publique $\mathbf{A} \in \mathcal{U}(\mathbb{M}^{m\times n})$).

Enc(s, A, m) : pour chiffrer un message $m \in \{0,1\}^m$, renvoie le chiffré $\mathbf{c} = \mathbf{A}\cdot\mathbf{s} + \mathbf{e} + \frac{q}{2}\mathbf{m} \bmod q$, en tirant indépendamment les vecteurs $\mathbf{e} \leftarrow \mathcal{U}([-q/4+1, q/4-1] \cap \mathbb{Z})$.

Dec(s, A, c) : Calcule $\mathbf{m}' = \mathbf{c} - \mathbf{A}\cdot\mathbf{s}$ et pour chaque élément du vecteur $\mathbf{m}'$, le mettre à 1 s'il est plus proche de $\frac{q}{2}$ que des extrémités 0 et $q-1$, et le mettre à 0 sinon.

1. Implémentez les fonctions
   - `int LWE_KG(long s[], long A[][]);`
   - `int LWE_enc(int c[], const long s[], const long A[][], const int m[]);`
   - `int LWE_dec(int m[], const long s[], const long A[][], const int c[]).`

   Correspondantes aux blocs en questions, en utilisant la multiplication simple-thread de l'exercice précédent. Les paramètres m, n, k seront des variables globales.

2. Utilisez votre chiffrement « monobloc » pour écrire la fonction qui implémente le mode CTR pour le chiffrement $\mathbf{LWE}_{\text{sym}}$: `int LWE_CTR(int c[][], const int s[], const int A[], const int m[][], const int IV)`.
   Votre implémentation utilisera un thread par bloc à chiffrer.

3. Chiffrez un message pour vos TDmen, et donnez la clef et le vecteur d'initialisation utilisés dans votre rapport.